Symbol Tables

- A **symbol table** is a major data structure used in a compiler:
  - Associates **attributes** with identifiers used in a program
  - For instance, a **type attribute** is usually associated with each identifier
  - A symbol table is a necessary component
    - Definition (declaration) of identifiers appears once in a program
    - Use of identifiers may appear in many places of the program text
  - Identifiers and attributes are entered by the analysis phases
    - When processing a definition (declaration) of an identifier
    - In simple languages with only global variables and implicit declarations:
      - The scanner can enter an identifier into a symbol table if it is not already there
    - In block-structured languages with scopes and explicit declarations:
      - The parser and/or semantic analyzer enter identifiers and corresponding attributes
  - Symbol table information is used by the analysis and synthesis phases
    - To verify that used identifiers have been defined (declared)
    - To verify that expressions and assignments are semantically correct – **type checking**
    - To generate intermediate or target code

Symbol Table Interface

- The basic operations defined on a symbol table include:
  - **allocate** – to allocate a new empty symbol table
  - **free** – to remove all entries and free the storage of a symbol table
  - **insert** – to insert a name in a symbol table and return a pointer to its entry
  - **lookup** – to search for a name and return a pointer to its entry
  - **set_attribute** – to associate an attribute with a given entry
  - **get_attribute** – to get an attribute associated with a given entry
- Other operations can be added depending on requirement
  - For example, a **delete** operation removes a name previously inserted
    - Some identifiers become invisible (out of scope) after exiting a block
- This interface provides an abstract view of a symbol table
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

Basic Implementation Techniques

- First consideration is how to **insert** and **lookup** names
- Variety of implementation techniques
- **Unordered List**
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problem of a fixed size array
  - Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average
- **Ordered List**
  - If an array is sorted, it can be searched using binary search – $O(\log_2 n)$
  - Insertion into a sorted array is expensive – $O(n)$ on average
  - Useful when set of names is known in advance – table of reserved words
- **Binary Search Tree**
  - Can grow dynamically
  - Insertion and lookup are $O(\log_2 n)$ on average

Hash Tables and Hash Functions

- A **hash table** is an array with index range: 0 to *TableSize* – 1
- Most commonly used data structure to implement symbol tables
- Insertion and lookup can be made very fast – $O(1)$
- A **hash function** maps an identifier name into a table index
  - A hash function, $h(name)$, should depend solely on *name*
  - $h(name)$ should be computed quickly
  - h should be **uniform** and **randomizing** in distributing names
  - All table indices should be mapped with equal probability
  - Similar names should not cluster to the same table index

Hash Functions

- Hash functions can be defined in many ways . . .
- A string can be treated as a sequence of integer words
  - Several characters are fit into an integer word
  - Strings longer than one word are folded using exclusive-or or addition
  - Hash value is obtained by taking integer word modulo *TableSize*
- We can also compute a hash value character by character:
  - $h(name) = (c_0 + c_1 + \ldots + c_{n-1})$ **mod** *TableSize*, where n is *name* length
  - $h(name) = (c_0 * c_1 * \ldots * c_{n-1})$ **mod** *TableSize*
  - $h(name) = (c_{n-1} + \alpha\,(c_{n-2} + \ldots + \alpha\,(c_1 + \alpha\, c_0)))$ **mod** *TableSize*
  - $h(name) = (c_0 * c_{n-1} * n)$ **mod** *TableSize*

Implementing a Hash Function

```
// Hash string s
// Hash value = (s_{n-1} + 16(s_{n-2} + .. + 16(s_1+16s_0)))
// Return hash value (independent of table size)

unsigned hash(char* s) {
  unsigned  hval = 0;
  while (*s != '\0') {
    hval = (hval << 4) + *s;
    s++;
  }
  return hval;
}
```

Another Hash Function

```
// Treat string s as an array of unsigned integers
// Fold array into an unsigned integer using addition
// Return hash value (independent of table size)
unsigned hash(char* s) {
  unsigned  hval = 0;
  while (s[0]!=0 && s[1]!=0 && s[2]!=0 && s[3]!=0){
    unsigned u = *((unsigned*) s);
    hval += u;  s += 4;
  }
  if (s[0] == 0) return hval;
  hval += s[0];
  if (s[1] == 0) return hval;
  hval += s[1]<<8;
  if (s[2] == 0) return hval;
  hval += s[2]<<16;
  return hval;
}
```

Last 3 characters are handled in a special way

Resolving Collisions – Open Addressing

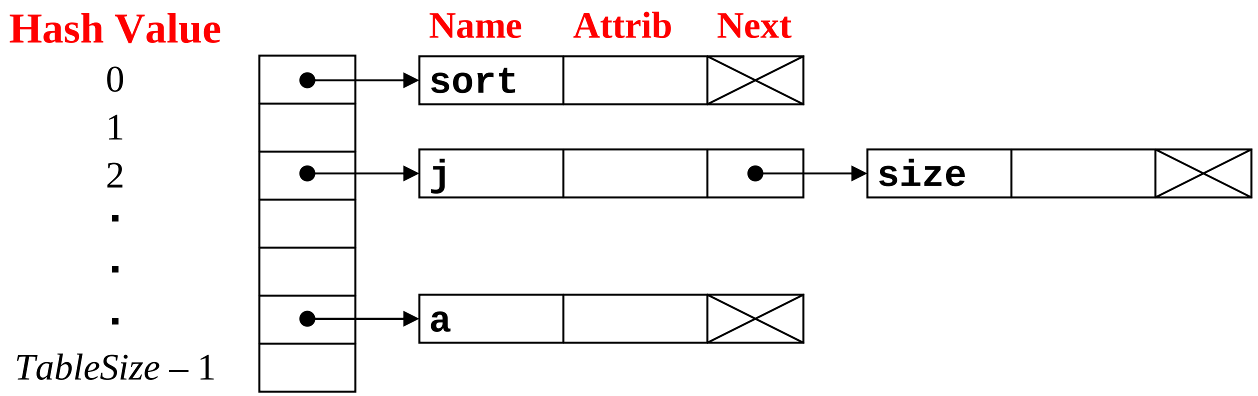
- A **collision** occurs when $h(name_1) = h(name_2)$ and $name_1 \neq name_2$
- Collisions are inevitable because
  - The name space of identifiers is much larger than the table size
- How to deal with collisions?
  - If entry $h(name)$ is occupied, try $h_2(name)$, $h_3(name)$, etc.
  - This approach is called **open addressing**
  - $h_2(name)$ can be $h(name) + 1$ **mod** *TableSize*
  - $h_3(name)$ can be $h(name) + 2$ **mod** *TableSize*

linear probing

Hash Value	Name	Attributes
0	`sort`	
1		
2	`size`	
.	`j`	
.	`a`	
TableSize – 1		

Chaining by Separate Lists

- Drawbacks of open addressing:
  - As the array fills, collisions become more frequent – reduced performance
  - Table size is an issue – dynamically increasing the table size is a difficulty
- An alternative to open addressing is **chaining by separate lists**
  - The hash table is an array of pointers to linked lists called **buckets**
  - Collisions are resolved by inserting a new identifier into a linked list
  - Number of identifiers is no longer restricted to table size
  - Lookup is $O(n/TableSize)$ when number of identifiers exceeds *TableSize*



Symbol Class Definition

```
class Symbol {              // Symbol class definition
friend class Table;         // To access private members
public:
  Symbol(char* s);          // Initialize symbol with name s
  ~Symbol();                // Delete name and clear pointers
  const char* id();         // Return pointer to symbol name
  Symbol* nextinlist();     // Next symbol in list
  Symbol* nextinbucket();   // Next symbol in bucket
  . . .                     // Other methods
private:
  char*   name;             // Symbol name
  Symbol* list;             // Next symbol in list
  Symbol* next;             // Next symbol in bucket
  . . .                     // Attributes (added later)
};
```

Symbol Class Implementation

```
// Initialize symbol and copy s
Symbol::Symbol(char* s) {
  name = new char[strlen(s)+1];
  strcpy(name,s);
  next = list = 0;
}

// Delete name and clear pointers
Symbol::~Symbol() {
  delete [] name;
  name = 0;
  next = list = 0;
}

const char* Symbol::id()          {return name;}

Symbol* Symbol::nextinbucket() {return next;}

Symbol* Symbol::nextinlist()    {return list;}
```

Symbol Table Class Definition

```
const unsigned HT_SIZE = 1021;       // Hash Table Size

class Table {                        // Symbol Table class
public:
  Table();                           // Initialize table
  Symbol* clear();                   // Clear symbol table
  Symbol* lookup(char*s);            // Lookup name s
  Symbol* lookup(char*s,unsigned h); // Lookup s with hash h
  Symbol* insert(char*s,unsigned h); // Insert s with hash h
  Symbol* lookupInsert(char*s);      // Lookup and insert s
  Symbol* symlist() {return first;}  // List of symbols
  unsigned symbols(){return count;}  // Symbol count
  . . .                              // Other methods
private:
  Symbol*  ht[HT_SIZE];              // Hash table
  Symbol*  first;                    // First inserted symbol
  Symbol*  last;                     // Last inserted symbol
  unsigned count;                    // Symbol count
};
```


Initialize and Clear a Symbol Table

```
// Initialize a symbol table

Table::Table() {
  for (int i=0; i<HT_SIZE; i++) ht[i] = 0;
  first = last = 0;
  count = 0;
}

// Clear a symbol table and return its symbol list

Symbol* Table::clear() {
  Symbol* list = first;
  for (int i=0; i<HT_SIZE; i++) ht[i] = 0;
  first = last = 0;
  count = 0;
  return list;
}
```

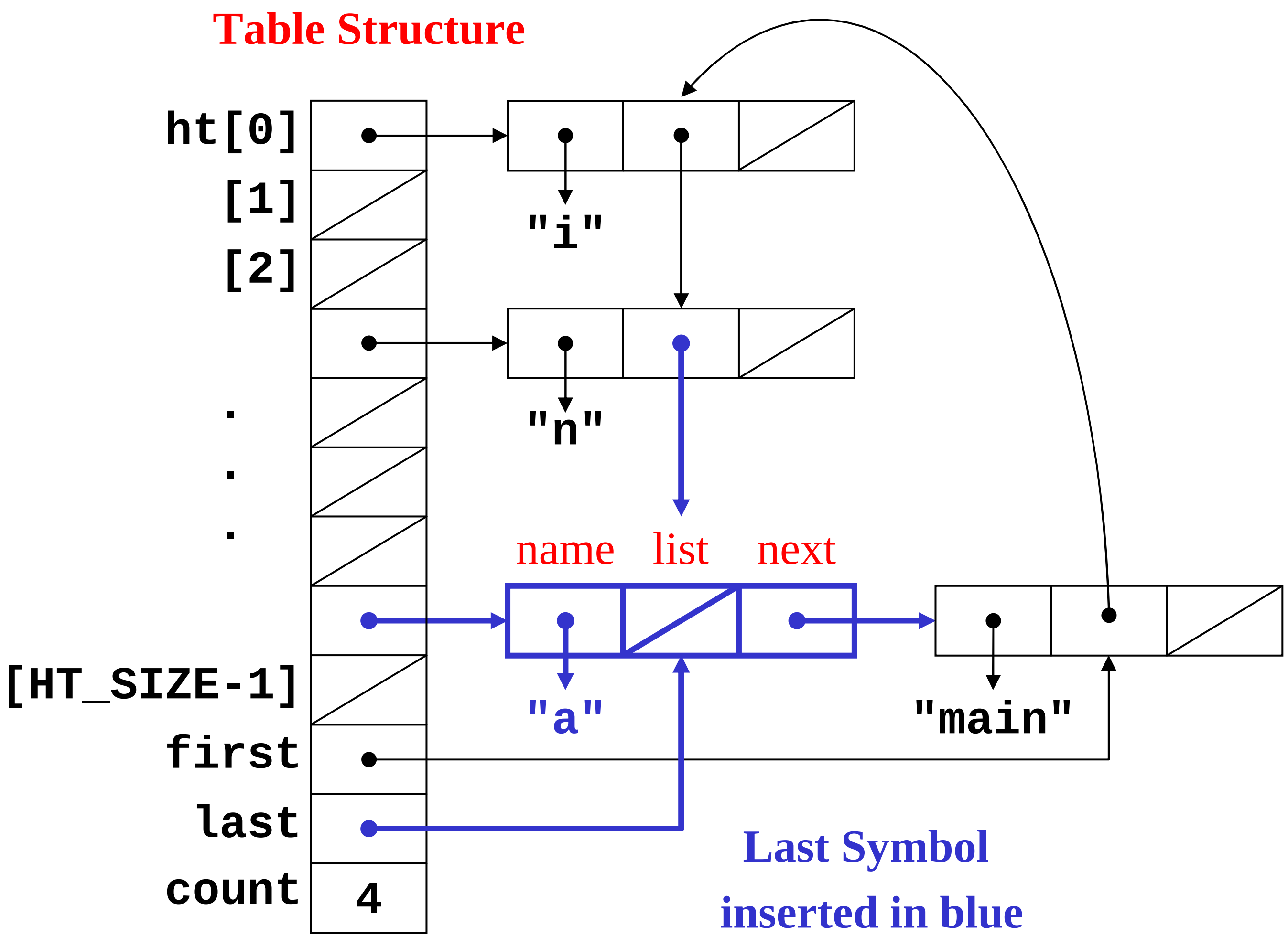
Lookup a Name in a Symbol Table

```
// Lookup name s in symbol table
// Return pointer to found symbol
// Return NULL if symbol not found

Symbol* Table::lookup(char* s) {
  unsigned h = hash(s);
  return lookup(s,h);
}

// Lookup name s with hash value h
// Hash value is passed to avoid its computation

Symbol* Table::lookup(char* s, unsigned h) {
  unsigned index = h % HT_SIZE;
  Symbol* sym = ht[index];
  while (sym != 0) {
    if (strcmp(sym->name, s) == 0) break;
    sym = sym->next;
  }
  return sym;
}
```

Insert a Name into a Symbol Table

```
// Insert name s with a given hash value h
// New symbol is allocated
// New symbol is inserted at front of a bucket list
// New symbol is also linked at end of symbol list in table
// Return pointer to newly allocated symbol

Symbol* Table::insert(char* s, unsigned h) {
  unsigned index = h % HT_SIZE;
  Symbol* sym = new Symbol(s);
  sym->next = ht[index];
  ht[index] = sym;
  if (count == 0) { first = last = sym; }
  else {
    last -> list = sym;
    last = sym;
  }
  count++;
  return sym;
}
```

Illustrating Symbol Insertion



Lookup and then Insert a Name

```
// Lookup first and then Insert name s
// If name s exists then return pointer to its symbol
// Otherwise, insert a new symbol and copy name s
// Return address of newly added symbol

Symbol* Table::lookupInsert(char* s) {
  unsigned h = hash(s);     // Computed once
  Symbol* sym;
  sym = lookup(s,h);        // Locate symbol first
  if (sym == 0) {           // If not found
    sym = insert(s,h);      // Insert a new symbol
  }
  return sym;
}
```